Stabilization-Preserving Atomicity Refinement

Mikhail Nesterenko [†] and Anish Arora [‡]

[†] *Mathematics and Computer Science, Kent State University, Kent, OH 44242 USA; and*
[‡] *Department of Computer and Information Science, Ohio State University, Columbus, OH 43210 USA.*

E-mail: mikhail@mcs.kent.edu; anish@cis.ohio-state.edu

Program refinements from an abstract to a concrete model empower designers to reason effectively in the abstract and architects to implement effectively in the concrete. For refinements to be useful, they must not only preserve functionality properties but also dependability properties. In this paper, we focus our attention on refinements that preserve the dependability property of stabilization. Specifically, we present a stabilization-preserving refinement of atomicity from an abstract model where a process can atomically access the state of all its neighbors and update its own state, to a concrete model where a process can only atomically access the state of any one of its neighbors or atomically update its own state. Our refinement is sound and complete with respect to the computations admitted by the abstract model, and induces linear step complexity and constant synchronization delay in the computations admitted by the concrete model. It is based on a bounded-space, stabilizing dining philosophers program in the concrete model, and is readily extended to: (a) solve stabilization-preserving semantics refinement, (b) solve the stabilizing drinking philosophers problem, (c) solve fairness refinement problem, and (d) allow further refinement into a message-passing model.

Key Words: atomicity refinement; stabilization; fault-tolerance; concurrency.

1. INTRODUCTION

Motivation. Concurrent programming involves reasoning about the interleaved execution of multiple processes. On one hand, if the granularity of atomic (indivisible) actions of a concurrent program is assumed to be coarse, the number of possible interleavings is kept small and the program design is made simple. On the other hand, if the program is to be efficiently implemented, its atomic actions must be fine-grain. This motivates the need for refinements from *high-atomicity* programs to *low-atomicity* programs.

To be useful, atomicity refinements should preserve essential properties of the high-atomicity program, such as stabilization. A program is stabilizing with respect to a set of legitimate states if, starting from an arbitrary initial state, the program is guaranteed to reach a legitimate state and remain in legitimate states thereafter. Thus, a stabilizing program does not need to be initialized and it is able to recover from transient failures. In this paper we focus our attention on atomicity refinements that preserve stabilization.

As can be expected, atomicity refinement introduces new control positions that arise from partially executed actions of the high-atomicity program. These control positions yield new illegitimate states in the low-atomicity program that do not have any corresponding states in the high-atomicity program, whence the challenge in preserving stabilization.

Problem formulation and our solution. Specifically, the abstract model we consider is one where a process can atomically access the state of all its neighbors and update its own state. The concrete model is one where a process can only atomically access the state of any one of its neighbors or atomically update its own state. (We also address further refinement to a message-passing model.) In all models, concurrent execution of actions of processes is in *interleaving* semantics where only one atomic action may be executed at a time.

Of course, the straightforward division of each high-atomicity action into a sequence of low-atomicity actions does not suffice: the actions in this sequence can interleave with actions executed by other processes yielding computations that are not possible in high-atomicity model. A simple strategy for refinement, then, is to execute each sequence in a mutually exclusive manner. This strategy unfortunately suffers from the loss of concurrency, since no two processes can execute sequences concurrently even if these sequences operate on completely disjoint state spaces. We are therefore led to solving the problem of dining philosophers, which requires mutual exclusion only between "neighboring" processes, and thus allows more concurrency. Note that for our purposes the solution must be (a) low-atomicity and (b) stabilizing. In this connection, we note that there already exist several stabilizing mutual exclusion programs in the literature [5, 6, 11, 15, 18], but none is easily generalized to thus solve dining philosophers.

Properties and extensions of our solution. We evaluate our refinement in terms of its correctness and its efficiency. Correctness requires at least that our refinement be *sound*, i.e., that every computation of the refined low-atomicity program corresponds to some computation of the high-atomicity program.

Soundness of a refinement does not imply that the concrete program necessarily implements all computations admitted by the abstract program. Thus, sound refinements may exclude some efficient (or otherwise desirable) high-atomicity computations. Since real computer architectures usually efficiently execute only some subset of all possible high-level computations, soundness alone is inadequate if the refinement fails to capture the relevant computation subsets. This deficiency is overcome by the *completeness* property of our refinement. A refinement is complete if it provides a low-atomicity computation for every possible high-atomicity computation. We consider terminating as well as non-terminating computations.

Therefore, to demonstrate soundness and completeness or our refinement we have to prove that it is *fixpoint-* and *fairness-preserving.* Fixpoint preservation means that a terminating computation of the high-atomicity program corresponds only to a terminating computation of the low-atomicity program and vice versa. Fairness preservation means that a weakly fair computation of the low-atomicity program corresponds to a weakly fair computation of the high-atomicity program.

To analyze time efficiency of refinements, we use two metrics [19]: step complexity and synchronization delay. Step complexity is the average number of low-atomicity actions needed to simulate a high-atomicity action. Synchronization delay is the average number of causally related low-atomicity actions that have to be taken between two high-atomicity actions of two neighbor processes. The step complexity of the refined program is $O(n)$ where n is the maximum degree of a process in the system. The refined program exhibits constant ($O(1)$) synchronization delay.

It is well-known that bounding the state of stabilizing programs is often challenging [4]. Yet a stabilizing program needs bounded space to be implemented reasonably. Our refinement has bounded space complexity.

Other types of refinement. In atomicity refinement, assumptions about the execution model other than action atomicity remain the same. For instance, both the high- and low-atomicity programs may be executed in interleaving semantics. Alternatively, both programs can be executed in power-set semantics, where any number of processes may each execute an atomic action at a time, or in partial-order semantics.

By way of contrast, let us consider other types of refinement. In the case of *semantics refinement* the atomicity of the program is left the same but the semantics of concurrency in program execution is refined. For instance, a program in interleaving semantics may be refined to execute (with identical actions) in power-set semantics [3]. The program is more easily reasoned about in the former semantics, but more easily implemented in the latter.

In the case of *fairness refinement* the fairness assumptions about the program execution are relaxed. For instance, a program that assumes weak fairness is refined to execute in the model with no fairness assumptions. Again, reasoning is easier in the former and implementation is easier in the latter.

Extensions of our refinement. Our refinement is directly applicable even if the concrete model is generalized to have *power-set* semantics instead of interleaving semantics. It is readily modified to drop the assumption of weak fairness in the concrete model, as well as to refine programs into a message-passing model.

Since dining philosophers programs prohibit neighboring processes from executing their high-atomicity actions concurrently even in scenarios where the neighbors' actions do not interfere with each other, we are led to describing how our stabilizing low-atomicity dining philosophers program is extended to solve the drinking philosophers problem, which makes our refinement more efficient.

Comparison to related work. Gouda and Haddix propose a solution to the semantics refinement problem [12], that provides stabilization preserving refinement from interleaving to power-set semantics. Their solution also refines fairness. The atomicity of their execution model is similar to our high-atomicity model. Gouda and Haddix also propose atomicity refinement. Their atomicity refinement transforms a high-atomicity program into a model with atomic actions that are more coarse than what our refinement provides. Their transformation is not fixpoint-preserving and, therefore, not complete. Their atomicity refinement exhibits the synchronization delay of $O(d)$ where d is the diameter of the system. The step complexity of their refinement differs depending on the load. Under light load the step complexity is $O(n * d * p)$ where n, p are the degree of a process in the system and the number of processes in the system respectively. Under heavy load the step complexity is $O(n * d)$.

Mizuno and Nesterenko [17] propose a refinement from interleaving semantics to program semantics where the actions of different processes are allowed to overlap. Unfortunately, their solution uses infinite variables. Since it is hard to bound these variables in a stabilizing program [4] there is no easy way to implement their refinement. Also, compared with the (unbounded space) transformation from high-atomicity model into message-passing model presented in [16], our solution has bounded space complexity.

Independently from our work, Antonoiu and Srimani [1] develop a refinement from a high-atomicity model into a message-passing system model. They assume that the layer that implements the low-atomicity model on message-passing systems has stabilized. Therefore, their refinement is equivalent to ours in a sense that it transforms a program from our high-atomicity model to our low-atomicity model. Although their refinement is of bounded space complexity, this bound M depends on the number of nodes in the system and has to be rather large for the low-atomicity program to work efficiently. The synchronization delay of their refinement is $O(d)$. The step complexity is $O(n * p + n * p/M)$ under light load and $O(n + n/M)$ under heavy load.

The rest of the paper is organized as follows. We define the model, syntax, and semantics of the programs we use in Section 2. We then present a low-atomicity dining philosophers program and prove its correctness and stabilization properties in Section 3. Next, in Section 4, we demonstrate how a high-atomicity program is refined using our dining philosophers program, and show the relationship between the refined program and the original high-atomicity program in terms of soundness, completeness, and fixpoint- and fairness- preservation. We estimate the performance of our refinement and compare it with other refinements in Section 5. We summarize the contribution of this paper and discuss extensions of our work in Section 6.

2. MODEL, SYNTAX, AND SEMANTICS

Model. A *program* consists of a set of processes and a binary reflexive symmetric relation N between them. The processes are assumed to have unique identifiers 1 through p. Processes P_i and P_j are called *neighbor processes* iff $(P_i, P_j) \in N$. Each process consists of a set of variables, a set of parameters, and a set of guarded commands (GCs.)

Syntax of high-atomicity programs. The syntax of a process P_i has the form:

process P_i
par ⟨declarations⟩
var ⟨declarations⟩

*[
⟨guarded command⟩ []...[] ⟨guarded command⟩
]

Declarations is a comma-separated list of items, each of the form:

$$\langle \textit{list of names} \rangle : \langle \textit{domain} \rangle$$

A variable can be updated (written to) only by the process that contains the variable. A variable can be read either by the process that contains the variable or by a neighbor process. We refer to a variable v that belongs to process P_i as v_i.

A parameter is used to define a set of variables and a set of guarded commands as one parameterized variable and guarded command respectively. For example, let a process P_i have parameter j ranging over values 2, 5, and 9; then a parameterized variable $x.j$ defines a set of variables $\{x.j \mid j \in \{2,5,9\}\}$ and a parameterized guarded command $GC.j$ defines the set of GCs:

$$GC.(j := 2) \;[]\; GC.(j := 5) \;[]\; GC.(j := 9)$$

A guarded command has the syntax:

$$\langle guard \rangle \longrightarrow \langle command \rangle$$

A guard is a boolean expression containing local and neighbor variables. A command is a finite comma separated sequence of assignment statements and branching statements. An assignment statement can be simple or quantified. A quantified assignment statement has the form:

$$(\| \langle range \rangle : \langle assignments \rangle)$$

A range is a bound variable and the values it assumes. Assignments is a comma separated list of assignment statements containing the bound variable. Similar to parameterized GC, a quantified statement represents a set of assignment statements where each assignment statement is obtained by replacing every occurrence of the bound variable in the assignments by its instance from the specified range.

Syntax of low-atomicity programs. The syntax for the low-atomicity program is the same as for the high-atomicity program with the following restrictions. The variable declaration section of a process has the following syntax:

var
private ⟨declarations⟩
public ⟨declarations⟩

A variable declared as private can be read only by the process that contains this variable. A public variable can also be read by a neighbor processes. A guarded command can be either *synch* or *update*. A synch GC mentions the public variables of one neighbor process and local private variables only. An update GC mentions only local variables (private or public.)

Let v_i be a private variable of P_i and v_j a public variable of P_j. We say that v_i is an *image* of v_j if there is a synch guard of process P_i that is enabled when $v_i \neq v_j$ and which assigns $v_i := v_j$ and v_i is not updated otherwise. The variable whose value is copied to the image variable is called the *source* of the image.

Semantics. The high-atomicity and the low-atomicity programs have the same semantics (cf. [2]). An assignment of values to variables of all processes in the concurrent program is a *state* of this program. A GC whose guard is **true** at some state of the program is *enabled* at this state. A *computation* is a maximal fair sequence of steps such that for each state s_i the state s_{i+1} is obtained by executing the command of some GC that is enabled at s_i. The maximality of a computation means that no computation can be a proper prefix of another computation and the set of all computations is suffix-closed. That is a computation either terminates in a state where none of the GCs are enabled or the computation is infinite. The fairness of a computation means that if a computation is infinite and a GC is enabled in all but finitely many states of the computation then this GC is executed infinitely often. That is, we assume *weak fairness* for command execution. A boolean variable is *set* in some state s if the value of this variable is **true** in s, otherwise the variable is *cleared* in s.

A *state predicate* (or just predicate) is a boolean expression over program variables. A state *conforms* to a predicate if this predicate evaluates to **true** in this state; otherwise, the state *violates* the predicate. By this definition every state conforms to predicate **true** and none conforms to **false**.

Let $\mathcal{P}$ be a program and R and S be state predicates on the states of $\mathcal{P}$. R is *closed* if every state of the computation of $\mathcal{P}$ that starts in a state conforming to R also conforms to R. R *converges* to S in $\mathcal{P}$ if R is closed in $\mathcal{P}$, S is closed in $\mathcal{P}$, and any computation starting from a state conforming to R contains a state conforming to S. $\mathcal{P}$ stabilizes to R iff **true** converges to R in $\mathcal{P}$. In the rest of the paper we omit the name of the program whenever it is clear from the context.

3. DINING PHILOSOPHERS PROGRAM

3.1. Description

Problem definition. The dining philosophers problem was first stated in [9]. Any process in the system can request access to a certain portion of code called *critical section*(CS). The objective of the program is to ensure that the following two properties hold:

safety - no two neighbor processes have guarded commands that execute CS enabled in one state;

liveness - a process requesting to execute CS is eventually allowed to do so.

process P_i
par $j : (P_i, P_j) \in N$
var
  public
    $ready_i$: boolean,
    $a_i.j, c_i.j$: (0..3)
  private
    $\texttt{request}_i$: boolean,
    $r_i.j, y_i.j$: boolean,
    $b_i.j, d_i.j$: (0..3)

$*[$

(dp1) $\texttt{request}_i \wedge \neg ready_i \wedge (\forall k : a_i.k = d_i.k) \wedge (\forall k > i : \neg y_i.k) \longrightarrow$
    $ready_i :=$ **true**,
    $(\| k > i : y_i.k := r_i.k,\ a_i.k := (a_i.k + 1)$ **mod** $4)$

$[]$

(dp2) $ready_i \wedge (\forall k : a_i.k = d_i.k) \wedge (\forall k < i : \neg r_i.k) \longrightarrow$
    / * critical section * /
    $ready_i :=$ **false**,
    $(\| k < i : a_i.k := (a_i.k + 1)$ **mod** $4)$

$[]$

(dp3) $c_i.j \neq b_i.j \longrightarrow$
    $c_i.j := b_i.j$

$[]$

(dp4) $r_i.j \neq ready_j \vee (b_i.j \neq a_j.i) \vee (d_i.j \neq c_j.i) \vee (j > i \wedge \neg ready_j \wedge y_i.j) \longrightarrow$
    $r_i.j := ready_j$,
    $b_i.j := a_j.i$,
    $d_i.j := c_j.i$,
    if $j > i \wedge \neg ready_j \wedge y_i.j$ **then** $y_i.j :=$ **false** **fi**

$]$

FIG. 1. Dining philosophers process

Design ideas. This section describes a program $\mathcal{DP}$ that solves the dining philosophers problem. The program is based on the followoing two ideas.

Prioritizing. To guarantee safety, if more than one neighbor process is in CS contention, only the one with the lowest identifier proceeds. To ensure liveness, when a process joins CS contention it records every processes already in CS contention with id greater than its own; after the process exits CS, it does not request CS again until the recorded neighbors enter CS.

Synchronization. A process indicates that it is in CS contention by setting a boolean variable *ready*. A process needs to syncrhonize with the neighbors in the following two cases: (a) to implement prioritizing a process executes CS only when *ready* of each neighbor with lower identifier is cleared; (b) similarly, a process P_i

requests CS only after every process with a lower identifier removes the record of P_i's previous CS contention. The latter is necessary to ensure that a process with a lower identifier eventually removes this record and the lower-id process is not denied joining CS contention indefinitely.

Note that in low-atomicity model CS execution is an update command. Thus, this command cannot directly read the variables of the neighbors and has to operate on the local images of these variables instead. Thus, a handshake mechanism is incorporated in $\mathcal{DP}$ to ensure that an image variable has the same value as a neighbor's *ready* when synchronization is needed. We describe the details of the handshake mechanism later in this section.

Variable and guarded command description. Every process P_i of $\mathcal{DP}$ is shown in Figure 1. To refer to a guarded command executed by some process we attach the process identifier to the name of the guarded command shown in Figure 1. For example, guarded command $dp1_i$ sets variable $ready_i$. We sometimes use identifiers of GCs in state predicates. For example, $dp1_i$ used in a predicate means that the guard of this GC is enabled.

Every P_i has the following variables. Private variables have one-letter names and public variables are given longer names.

- $\texttt{request}_i$ - captures the reaction of the environment. It is a read-only variable which is used in program composition in later sections. P_i *wants to enter* its CS if $\texttt{request}_i$ is set.
- $ready_i$ - indicates if P_i tries to execute its CS. P_i is *in CS contention* if $ready_i$ is set.
- $r_i.j$ - records whether P_j is in CS contention, it is an image of $ready_j$.
- $y_i.j$ - records if P_j requests CS and needs to be allowed to access it before P_i can itself request CS again. It is maintained for each P_j such that $j > i$; it is called *yield variable.*
- $a_i.j, b_i.j, c_i.j, d_i.j$ - used for synchronization between neighbor processes; they are called *handshake variables.*

A process has a sequence of handshake variables with each neighbor. For example (see Fig. 2), process P_i and its neighbor P_j have the following sequence: $H_{ij} = \langle a_i.j$, $b_j.i$, $c_j.i$, $d_i.j\rangle$. Note that P_j has a similar sequence with P_i: $H_{ji} = \langle a_j.i$, $b_i.j$, $c_i.j$, $d_j.i\rangle$. We say that $a_i.j$ has a token if $a_i.j$ is equal to $d_i.j$. We say that any of the other variables has a token if it is *not* equal to the variable preceding it in H_{ij}. $\mathcal{DP}$ is designed such that H_{ij} forms a ring similar to the a ring used in K-state stabilizing protocol described in [10]. Process P_i starts the circulation of the token by incrementing $a_i.j$.

Every P_i has the following four GCs. Commands $dp1_i$, $dp2_i$, and $dp3_i$ are update and $dp4_i$ is synch.

$dp1_i$ - enabled when P_i wants to enter CS, P_i is not in CS contention, for every neighbor P_j, $a_i.j$ has a token, and yield variables for processes with identifiers greater than i are not set. The command sets $ready_i$ joining CS contention, it sets yield variables for processes who are in CS contention, and increments $a_i.j$ for every

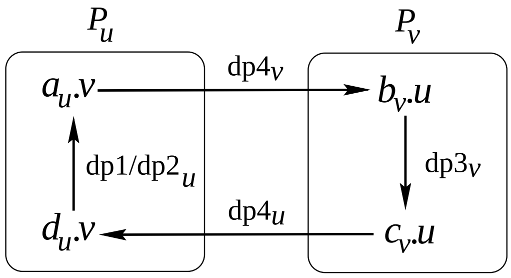

FIG. 2. Neighbor synchronization using handshake variables

P_j with identifier greater than i passing the token in the handshake sequence. Note that when $a_i.j$ collects the token again P_j is informed of P_i's joining CS contention (that is $r_j.i$ is set.)

$dp2_i$ - enabled when P_i is in CS contention, every $a_i.j$ has the token, and processes with smaller identifiers are not in CS contention. The command clears $ready_i$ and increments $a_i.j$ for every P_j with identifier less than i passing the tokens again. Note that when the tokens are collected, every neighbor P_j is informed that P_i exited CS and yield variable $y_j.i$ is cleared.

$dp3_i$ - enabled when $c_i.j$ has the token. The command sets $c_i.j$ equal to $b_i.j$ thus passing the token from $c_i.j$ to $d_j.i$.

$dp4_i$ - enabled when either $b_i.j$ or $d_i.j$ has the token or when the image is not equal to $ready_j$ or when $ready_j$ is cleared but the corresponding yield variable is set. The command passes the tokens from $b_i.j$ to $c_i.j$ and from $d_i.j$ to $a_i.j$. It also copies the value of $ready_j$ to it's image $r_i.j$ and clears the yield variable $y_i.j$ when P_j is not in CS contention.

Handshake principle and example computation. We demonstrate the operation of handshake on an example computation. Let us suppose that process P_u has a neighbor P_v and $u < v$. Suppose also that the computation starts from a state where P_v is in CS contention ($ready_v$ is set), P_u is not, and $a_u.v$ has the token. Let us assume that P_u joins CS contention by executing $dp1_u$. This command increments $a_u.v$ (which passes the token to $b_v.u$), sets $ready_u$, and $y_u.v$. Since $a_u.v$ is not equal to $b_v.u$, $dp4_v$ is enabled. When it is executed, the token is passed to $c_v.u$ and $r_v.u$ is set. Since $r_v.u$ is set, P_v cannot enter CS until P_u executes CS and clears $ready_u$. When $c_v.u$ has the token, $dp3_v$ is enabled. When it is executed, the token is passed on to $d_u.v$. This enables $dp4_u$. When $dp4_u$ is executed, the token returns to $a_u.v$. If P_u collects all tokens and there is no neighbor in CS contention with lower id, then P_u executes CS.

Note that $y_u.v$ is set. Therefore, P_u cannot enter CS again until P_v executes its CS and clears $ready_v$. When P_v executes CS, P_v circles to token along H_{vu}. This ensures that P_v clears $y_u.v$ before P_v can enter CS again.

3.2. Correctness of $\mathcal{DP}$

We prove the correctness of $\mathcal{DP}$ by first demonstrating that it stabilizes to a certain invariant. Then we show that this invariant guarantees safety and liveness properties of the dining philosophers problem.

3.2.1. Stabilization

Let P_u and P_v be any two neighbor processes.

PROPOSITION 3.1. $\mathcal{DP}$ stabilizes to the following predicate:

$$\text{there can be one and only one token in } H_{uv} \qquad (R_1)$$

This proposition is proven in the Appendix A.1.

Lemma 3.1 states sufficient conditions for the image $r_v.u$ to contain the same value as the source.

LEMMA 3.1. $\mathcal{DP}$ stabilizes to the following predicates:

$$((u < v) \wedge (a_u.v = b_v.u) \wedge ready_u) \Rightarrow r_v.u \qquad (R_2)$$
$$((u > v) \wedge (a_u.v = b_v.u) \wedge \neg ready_u) \Rightarrow \neg r_v.u \qquad (R_3)$$

Proof. By Proposition 3.1, $\mathcal{DP}$ stabilizes to R_1. To prove the lemma we need to demonstrate that R_2 and R_3 are closed when R_1 holds and that R_1 converges to R_2 and R_3. We prove convergence and closure for R_2 only. The stabilization of R_3 can be proven similarly.

We show closure first. Out of the eight GCs of processes P_u and P_v only $dp1_u$, $dp2_u$ and $dp4_v$ affect the variables of the predicate. Command $dp1_u$ is executed only when $a_u.v = d_u.v$. That is when variable $a_u.v$ has the token. Since R_1 holds there can be no other tokens in the handshake variables. Thus, $a_u.v = b_v.u$ when $dp1_u$ is executed. Therefore $dp1_u$ sets the antecedent of our predicate to **false** by clearing $ready_u$. Command $dp2_u$ also sets the antecedent of our predicate to **false**. If $dp4_v$ sets the antecedent to **true** (by setting $a_u.v = b_v.u$), then the consequent is also set to **true** (since $r_v.u = ready_u$ after the execution of $dp4_v$).

To demonstrate convergence we note that when R_2 does not hold $dp4_v$ is enabled. When $dp4_v$ is executed the predicate holds. ■

Lemma 3.2 states sufficient conditions for a yield variable to be cleared.

LEMMA 3.2. $\mathcal{DP}$ stabilizes to the following predicate:

$$((u > v) \wedge (a_u.v = b_v.u) \wedge \neg ready_u) \Rightarrow \neg y_v.u \qquad (R_4)$$

Proof. We show that R_4 is closed when R_1 and R_3 hold and that R_4 converges.

We show closure first. The guarded commands that affect our predicate are: $dp1_u$, $dp2_u$, $dp1_v$ and $dp4_v$. $dp1_u$ and $dp2_u$ set the antecedent to **false** and therefore do not violate the predicate. $dp1_v$ affects only the consequent of our predicate. By R_3 when the antecedent of our predicate is **true**, then $r_v.u =$ **false**. Therefore, when the antecedent is **true** and $dp1_v$ is executed, variable $y_v.u$ remains **false** and our predicate is not violated.

$dp4_v$ can set the consequent of our predicate to **true** only. Also, if $dp4_v$ sets the antecedent of our predicate to **true** (by setting $a_u.v = b_v.u$ while $ready_u =$ **false**) the consequent of our predicate must also be **true** after the execution of $dp4_v$ (if $ready_u$ is cleared before the execution of $dp4_v$, $y_v.u$ is set to **false** by $dp4_v$).

Similar to Lemma 3.1 we demonstrate convergence by pointing out that when our predicate does not hold $dp4_v$ is enabled. When $dp4_v$ is executed the predicate holds. ■

We now define a predicate $I_{\mathcal{DP}}$ (which stands for invariant of $\mathcal{DP}$) such that every computation of $\mathcal{DP}$ that starts at a state conforming to $I_{\mathcal{DP}}$ satisfies safety and liveness. $I_{\mathcal{DP}}$ is: for every pair of neighbor processes $R_1 \wedge R_2 \wedge R_3 \wedge R_4$. In other words, in every state conforming to $I_{\mathcal{DP}}$, every pair of neighbor processes conforms to each predicate in the above list.

THEOREM 3.1. $\mathcal{DP}$ stabilizes to $I_{\mathcal{DP}}$.

Thus every execution of $\mathcal{DP}$ eventually reaches a state conforming to $I_{\mathcal{DP}}$. In the next two subsections we show that every computation that starts from a state conforming to $I_{\mathcal{DP}}$ satisfies safety and liveness properties.

3.2.2. Safety

THEOREM 3.2 (Safety). In a state conforming to $I_{\mathcal{DP}}$, no two neighbor processes have their guarded commands that execute critical section enabled.

Proof. Let us assume that P_u and P_v are neighbors and $u < v$. If $dp2_u$ is enabled then $a_u.v = d_u.v$. By R_1 this means that $a_u.v = b_v.u$. If $dp2_u$ is enabled then $ready_u$ is set. Therefore, byR_2: $r_v.u$ is also set. When $r_v.u$ is set $dp2_v$ cannot be enabled. ■

3.2.3. Liveness

For a process P_u and its neighbor P_v the value of the variable $a_u.v$ is changed only when all a variables of process P_u have their tokens. The following observation can be made on the basis of Proposition 3.1.

PROPOSITION 3.2. All a variables of a process eventually get the tokens. That is a state conforming to: $\exists v : (P_v, P_u) \in N : a_u.v \neq d_u.v$ is eventually followed by a state where: $\forall v : (P_v, P_u) \in N : a_u.v = d_u.v$

LEMMA 3.3. If a process P_u is in CS contention it is eventually allowed to execute CS.

Proof. To prove the lemma we need to show that for any P_u, if $ready_u$ is set then $dp2_u$ eventually gets enabled, stays enabled and gets executed. The proof is by induction on the process identifiers.

Suppose process P_1 has $ready_1$ set in some state of a computation. By Proposition 3.2 all tokens are eventually collected at a_us variables and $dp2_1$ gets enabled.

When $ready_1$ is set the only command that can manipulate the tokens is $dp2_1$. Therefore, a_us do not give up the tokens unless $dp2_1$ is executed and $dp2_1$ stays enabled until executed. Thus, the lemma holds for P_1.

Suppose now that the lemma holds for processes with identifiers smaller than u and P_u has $ready_u$ set at some state of a computation. Again, by Proposition 3.2, for any neighbor P_v, $a_u.v$ gets the token. To demonstrate that $dp2_u$ eventually becomes and stays enabled until it is executed we need to show that for any neighbor P_v such that $v < u$, $r_u.v$ is eventually cleared and never set until $dp2_u$ is executed. There are two cases:

- $ready_v$ is set in infinitely many states of the computation. By assumption the lemma holds for P_v. Therefore, every such state is eventually followed by a state where $ready_v$ is cleared. After such a state $ready_v$ is set again. Thus $dp1_v$ and $dp2_v$ are executed infinitely many times during the computation. If $ready_u$ is set, eventually $r_v.u$ is set as well. When $dp1_v$ is executed in a state where $r_v.u$ set, this command sets $y_v.u$. $y_v.u$ is not cleared while $ready_u$ (and subsequently $r_v.u$) is set. When $y_v.u$ is set $dp1_v$ cannot be executed and $ready_v$ remains cleared while $ready_u$ is set. If $ready_v$ is cleared, eventually $r_u.v$ is cleared as well. Thus, $r_u.v$ stays cleared until $dp2_u$ is executed.
- $ready_v$ is set in only finitely many states of the computation. In this case there is a suffix of the computation where $ready_v$ is not set in any of the states. Thus eventually $r_u.v$ is cleared and remains cleared for the rest of the computation.

Thus $dp2_u$ becomes enabled, stays enabled, and gets executed. ■

LEMMA 3.4. If a process P_u wants to enter CS it eventually joins CS contention.

Proof. To prove the lemma we need to show that if $\texttt{request}_u$ is set then $dp1_u$ (that sets $ready_u$) is eventually executed. Let us assume that $\texttt{request}_u$ is set and it is not cleared at least until $ready_u$ is set. By Proposition 3.2 for any P_u's neighbor P_v, $a_u.v$ eventually gets the token. When $ready_u$ is cleared $a_u.v$ ever gives up the token unless $dp1_u$ is executed. Therefore, to demonstrate that the $dp1_u$ gets enabled we need to show that for any neighbor P_v such that $v > u$, $y_u.v$ eventually gets cleared. Note that $y_u.v$ is set only when $dp1_u$ is executed.

There can be only two cases:

- $ready_v$ is set in infinitely many states. By Lemma 3.3 this implies that $ready_v$ is also cleared in infinitely many states as well. Therefore, $dp1_v$ is executed infinitely many times. By Predicate R_4, $dp1_v$ can be executed only in a state where $y_u.v$ is cleared.
- $ready_v$ is set in only finitely many states. In this case there is a suffix of the execution where $ready_v$ is never set. If $y_u.v$ is set then $dp4_u$ is enabled. When $dp4_u$ is executed $y_u.v$ is cleared.

■

The following theorem unifies Lemmas 3.3 and 3.4.

THEOREM 3.3 (Liveness). If $I_{\mathcal{DP}}$ holds, a process that wants to enter CS is eventually allowed to do so.

process P_i
var x_i

$*[$

(h1) $g_i(x_i,\ \langle x_k \mid (P_i, P_k) \in N\rangle) \longrightarrow x_i := f_i(x_i,\ \langle x_k \mid (P_i, P_k) \in N\rangle)$

$]$

FIG. 3. High-atomicity process

4. THE REFINEMENT

4.1. High-Atomicity Program

Each process P_i of high-atomicity program ($\mathcal{H}$) is shown in Figure 3. To simplify the presentation we assume that P_i contains only one GC. We provide the generalization to multiple GCs later in the section. Each P_i of $\mathcal{H}$ contains a variable x_i which is updated by $h1_i$. The type of x_i is arbitrary. The guard of this GC is a predicate g_i that depends on the values of x_i and variables of neighbor processes. The command of $h1_i$ assigns a new value to x_i. The value is supplied by a function f_i which again depends on the previous value of x_i as well as on the values of the variables of the neighbors. Recall, that unlike low-atomicity program such as $\mathcal{DP}$, a GC of $\mathcal{H}$ can read variables of neighbor processes and update its own variables in one GC.

4.2. Refining $\mathcal{H}$

Problem definition. *Stuttering* is a sequence of identical states. A computation of a program $\mathcal{P}$ maps to a computation of $\mathcal{Q}$ if the sequence of states of the computation of $\mathcal{P}$ maps to the sequence of states of $\mathcal{Q}$ such that this sequence is the computation of $\mathcal{Q}$ if finite stuttering is eliminated. Note that the computation mapping does not eliminate an infinite sequence of identical states.

Given a high-atomicity program $\mathcal{H}$, the refinement consists of a low-atomicity program $\mathcal{C}$ and a non-trivial mapping from the states of $\mathcal{C}$ to the states of $\mathcal{H}$ such that the following two properties hold:

soundness every computation of $\mathcal{C}$ maps to a compuation of $\mathcal{H}$;

completeness for every computation of $\mathcal{H}$ there is a computation of $\mathcal{C}$ that maps to the computation of $\mathcal{H}$.

Refinement idea. Unlike a guarded command of $\mathcal{H}$, the GC of $\mathcal{C}$ cannot directly read the variables of the neighbor process. Thus, the GC that corresponds to the high-atomicity command has to operate on the images of these variables. To ensure soundness $\mathcal{C}$ has to guarantee that (a) when a process executes a command that corresponds to a high-atomicity GCs, no neighbor process can execute such a command and (b) when such a command is executed it reads the up-to-date images of neighbors' variables. $\mathcal{C}$ uses $\mathcal{DP}$ to provide this guarantee.

Superposition. To produce the refinement $\mathcal{C}$ of $\mathcal{H}$ we *superpose* additional commands on the GCs of $\mathcal{DP}$. $\mathcal{C}$ consists of $\mathcal{DP}$, superposition variables, superposition commands and superposition GCs. The superposition variables are disjoint from variables of $\mathcal{DP}$. Each superposition command has the following form:

$$\langle GC\ of\ \mathcal{DP}\rangle \parallel \langle command\rangle$$

The type of combined GC (synch or update) is the same as the type of the GC of $\mathcal{DP}$. The superposition commands and GCs can read but cannot update the variables of $\mathcal{DP}$. They can update the superposed variables. Operationally speaking a superposed command executes in parallel (synchronously) with the GC of $\mathcal{DP}$ it is based upon, and a superposed GC executes independently (asynchronously) of the other GCs. Superposition preserves liveness and safety properties of the underlying program ($\mathcal{DP}$). In particular, if $\mathcal{DP}$ stabilizes to R, so does $\mathcal{C}$. Thus, $I_{\mathcal{DP}}$ is also an invariant of $\mathcal{C}$. Refer to [8] for more details on superposition.

process P_i
par $j : (P_i, P_j) \in N$
var
  public x_i
  private
    $x_i.j$,
    $\texttt{request}_i$: boolean
$*[$
(c1) $dp1$
$[]$
(c2) $dp2 \parallel \left(\begin{array}{l} \textbf{if}\ \ g_i(x_i, \langle x_i.k \mid (P_i,P_k) \in N\rangle)\ \ \textbf{then} \\ \quad x_i := f_i(x_i,\ \langle x_i.k \mid (P_i,P_k) \in N\rangle) \\ \textbf{fi}, \\ \texttt{request}_i := \textbf{false} \end{array} \right)$
$[]$
(c3) $dp3$
$[]$
(c4) $dp4 \parallel (\ \textbf{if}\ \ x_i.j \neq x.j\ \textbf{then}\ \ x_i.j := x.j,\ \texttt{request}_i := \textbf{true}\ \ \textbf{fi})$
$[]$
(c5) $x_i.j \neq x.j \quad \longrightarrow \quad x_i.j := x.j,\ \texttt{request}_i := \textbf{true}$
$[]$
(c6) $g_i(x_i, \langle x_i.k \mid (P_i,P_k) \in N\rangle) \wedge \neg\texttt{request}_i \quad \longrightarrow$
    $\texttt{request}_i := \textbf{true}$
$]$

FIG. 4. Refined process

Variable and guarded command description. Each process P_i of the composed program ($\mathcal{C}$) is shown in Figure 4. For brevity, we only list the superposed variables in the variable declaration section. Besides the x_i we add $x_i.j$ which is an image of x_j for every neighbor P_j. Superposed variable $\texttt{request}_i$ is read by $\mathcal{DP}$. Yet it does not violate the liveness and safety properties of $\mathcal{DP}$ since no assumptions about this variable were made when the properties of $\mathcal{DP}$ were proven.

The GCs of $\mathcal{DP}$ are shown in abbreviated form. We superpose the execution of $h1$ on $dp2$. Note that $c2$ is an update GC. Therefore, the superposed command cannot read the value of x_j of a neighbor P_j directly as $h1$ does. The image $x_i.j$ is used instead. We superpose copying of the value of x_j into $x_i.j$ on $dp4$. Thus, the images of neighbor variables of $\mathcal{H}$ are equal to the sources when $h1$ is executed by $\mathcal{C}$. We add a superposition GC $c5$ that copies the value of x_j into $x_i.j$. This GC ensures that no deadlock occurs when an image is not equal to its source. Variable $\texttt{request}_i$ is set when one of the images of the superposed variables is found to be different from the sources or when the guard of $h1$ evaluates to **true** ($c6$). Variable $\texttt{request}_i$ is cleared after $h1$ is executed.

Component projection. Recall that a global state is an assignment of values to all the variables of a concurrent program. If a program is composed of several component programs, then a *component projection* of a global state s is a part of s consisting of the assignment of values to the variables used only in that program component. We define the projection of $\mathcal{C}$ onto $\mathcal{H}$ to the be the refinement mapping.

So far we assumed that $\mathcal{H}$ has only one GC. The refined program can be extended to multiple GCs. In this case, $c2$ has to select one of the enabled GCs of $\mathcal{H}$ and execute it. $c6$ has to be enabled when at least one of the GCs of $\mathcal{H}$ is enabled. We prove the correctness of $\mathcal{C}$ assuming that $\mathcal{H}$ has only one GC. In a straightforward manner, our argument can be extended to encompass multiple GCs.

4.3. Correctness of the Refinement

As we did for $\mathcal{DP}$, we demonstrate correctness of $\mathcal{C}$ by first proving that it stabilizes to a certain invariant. We then show that this invariant guarantees that $\mathcal{C}$ conforms to the properties of a refinement. Throughout this section we assume that P_u and P_v are neighbor processes.

4.3.1. Stabilization

The predicates in Lemmas 4.1 and 4.2 state the sufficient condition for the images of the neighbor variables of $\mathcal{H}$ to be equal to the sources.

LEMMA 4.1. $\mathcal{C}$ stabilizes to the following predicates:

$$((u < v) \wedge (a_u.v = d_u.v) \wedge ready_u) \Rightarrow (x_u.v = x_v) \qquad (R_5)$$

$$((u > v) \wedge \neg r_u.v) \Rightarrow (x_u.v = x_v) \qquad (R_6)$$

Proof. To demonstrate the stabilization of these predicates we show that they are closed under the assumption that $I_{\mathcal{DP}}$ holds and that they converge.

We show the closure of R_5 first. Of the twelve guarded commands of P_u and P_v, the following GCs affect R_5: $c1_u$, $c2_u$, $c4_u$, $c5_u$, and $c2_v$. $c1_u$ and $c2_u$ set the antecedent to **false**; $c4_u$ and $c5_u$ set the consequent to **true**. If $c2_v$ holds at a certain state then $\neg r_v.u$. By R_2 this implies that at this state the antecedent of R_5 is false. Therefore, the execution of $c2_v$ does not violate R_5.

To show the closure of R_6 we note that only $c4_u$, $c5_u$, and $c2_v$ affect R_6. Again both $c4_u$ and $c5_u$ set the consequent to **true**. Holding of $c2_v$ implies that the antecedent of R_6 is false by R_2.

To demonstrate convergence of both predicates we observe that when either of them does not hold $c5_u$ is enabled and it remains enabled until executed. After $c5_u$ is executed the predicates hold. ■

The following corollary can be deduced from the lemma.

COROLLARY 4.1. If $I_{\mathcal{DP}}$ holds, $c2$ is executed only when the images of the neighbor variables are equal to the sources. That is:

$$\forall (P_u, P_v) \in N : c2_u \Rightarrow (x_u.v = x.v)$$

LEMMA 4.2. $\mathcal{C}$ stabilizes to the following predicates:

$$((u < v) \wedge (a_u.v = b_v.u) \wedge \neg ready_u) \Rightarrow (x_u = x_v.u) \qquad (R_7)$$

$$((u > v) \wedge (a_u.v = b_v.u) \wedge ready_u) \Rightarrow (x_u = x_v.u) \qquad (R_8)$$

Proof. We prove the stabilization of R_7. The stabilization of R_8 can be shown likewise. Similar to Lemma 4.1 we demonstrate the stabilization of the R_7 by showing that it is closed under the assumption that $I_{\mathcal{DP}}$ holds and that it converges.

We show the closure first. Of the twelve guarded commands of P_u and P_v, the following GCs affect R_7: $c1_u$, $c2_u$, $c4_v$, $c5_v$. $c1_u$ and $c2_u$ set the antecedent to **false**; $c4_u$ and $c5_u$ set the consequent to **true**. Therefore, R_7 is not violated.

To demonstrate convergence we observe that when R_7 does not hold $c5_u$ is enabled and it remains enabled until executed. After $c5_u$ is executed the predicate holds. ■

Similar to $I_{\mathcal{DP}}$, we define the invariant for $\mathcal{C}$ (denoted $I_{\mathcal{C}}$) to be the conjunction of $I_{\mathcal{DP}}$, R_5, R_6, R_7, and R_8. On the basis of Theorem 3.1, Lemma 4.1, and Lemma 4.2 we can conclude:

THEOREM 4.1. $\mathcal{C}$ stabilizes to $I_{\mathcal{C}}$.

4.3.2. Soundness

It is not difficult to deduce that when $I_{\mathcal{C}}$ holds the component projection of a the sequence of states of $\mathcal{C}$ is a sequence of states of $\mathcal{H}$ produced by execution of GCs of $\mathcal{H}$. We demonstrate this as a part of the proof of Theorem 4.3. However, we still need to show that this sequence of states of $\mathcal{H}$ is a computation. For that we need to demonstrate that this sequence is maximal and fair. There are two cases to consider.

• The sequence of states is finite. A *fixpoint* is a state where none of the GCs of the program are enabled. For a finite sequence of states of $\mathcal{H}$ to be a computation it has to end in a fixpoint. Furthermore, the sequence has to correspond to a finite computation of $\mathcal{C}$. Theorem 4.2 demonstrates that.

• The sequence of states is infinite. In this case we need to show that weak fairness of GC execution is preserved. We show it in the proof of Theorem 4.3.

PROPOSITION 4.1. Let s be a fixpoint of $\mathcal{C}$. The following is true in s:

• $a_u.v = b_v.u = c_v.u = d_u.v$
• $r_u.v = ready_v$
• $ready_u$ is cleared;
• if $u < v$ then $y_u.v$ is cleared;
• $x_u.v = x_v$

THEOREM 4.2 (Fixpoint preservation). When $I_{\mathcal{C}}$ holds, a projection of a fixpoint of $\mathcal{C}$ is a fixpoint of $\mathcal{H}$; and if a computation of $\mathcal{C}$ starts from a state which projection is a fixpoint of $\mathcal{H}$ then this computation ends in a fixpoint.

Proof. Let s be a fixpoint of $\mathcal{C}$. By Proposition 4.1, at state s, $ready_u$ and $y_u.v$ (if $u < v$) are cleared, and $a_u.v$ has the token. Since s is a fixpoint, $c1_u$ is not enabled, therefore, $\texttt{request}_u$ is cleared at s.

Since $c6_u$ is not enabled and $\texttt{request}_u$ is cleared, $h1_u$ is not enabled either. By Proposition 4.1, $x_u.v$ is equal to x_v at s. Therefore the projection of s does not have $h1$ enabled and this projection is a fixpoint.

We now show that the computation that starts at a state s_1 such that the projection of s_1 is a fixpoint this computation ends in a fixpoint. By Corollary 4.1 if $I_{\mathcal{C}}$ holds, $x_u.v = x_v$ when $c2_u$ is executed. Thus, if the projection of the initial state of the computation is a fixpoint, $h1_u$ is not executed during this computation. Therefore the projection of every state of this computation is a fixpoint of $\mathcal{H}$.

Since the projection of every state is a fixpoint, eventually there is a state s_1 such that $x_u.v = x_v$. After this, if $\texttt{request}_u$ is cleared it is never set. Also, $c5_u$ and $c6_u$ cannot be enabled after s_1. If $\texttt{request}_u$ is set, by Theorem 3.3, $c2_u$ is eventually executed which clears $ready_u$ and $\texttt{request}_u$. After $\texttt{request}_u$ is cleared $c1_u$ cannot be enabled. Therefore $ready_u$ is cleared throughout the rest of the computation. Thus, $c2_u$ cannot enabled. If $c1_u$ and $c2_u$ are never executed then eventually $a_u.v = b_v.u = c_v.u = d_u.v$, $r_u.v = ready_v$ and if $u < v$ then $y_u.v$ is cleared. Thus $c3_u$ and $c4_u$ are disabled and $\mathcal{C}$ reaches a fixpoint. ■

Let σ_C and σ_H be computations of $\mathcal{C}$ and $\mathcal{H}$ respectively.

LEMMA 4.3 (Fairness preservation). If $I_{\mathcal{C}}$ holds and $h1_u$ is continually enabled in the projection of σ_C, then $h1_u$ is eventually executed in σ_C.

Proof. Let s be a state of σ_C such that $h1_u$ is enabled in the projection of s. If $\texttt{request}_u$ is set in s then (by Theorem 3.3) $c2_u$ is eventually executed. Let s_1 be the state when $c2_u$ is executed. By Corollary 4.1, $x_u.v = x_v$ at s_1. Then, since $h1_u$ is enabled in the projection of s_1 at it is also enabled at s_1. Thus, $h1_u$ is executed at s_1.

If $\texttt{request}_u$ is not set in s there can be two cases:

- every neighbor P_v executes $h1_v$ only finitely many times during σ_C. Let s_2 be the state after P_v executed $h1_v$ for the last time. If $x_u.v \neq x_v$ in s_2, then either $c4_u$ or $c5_u$ is eventually executed which sets $\texttt{request}_u$. This leads to eventual execution of $h1_u$.
- A neighbor P_v of P_u executes $h1_v$ infinitely many times. This implies that $c1_v$ and $c2_v$ are executed infinitely often. If $u < v$ by R_7, $x_v = x_u.v$ when $c1_v$ is enabled. Therefore P_u must execute either $c4_u$ or $c5_u$ infinitely often. Also, when $c4_u$ is executed $x_v \neq x_u.v$. Therefore, $\texttt{request}_u$ gets enabled which leads to eventual execution of $h1_u$.

Similar argument applies to the case of $u > v$ and R_8.

■

THEOREM 4.3 (Soundness). If a computation of $\mathcal{C}$, σ_C, starts at a state where $I_{\mathcal{C}}$ holds, then the projection of σ_C, σ_H, is a computation of $\mathcal{H}$.

Proof. By Corollary 4.1, when $c2_u$ is executed the images of the variables used in $\mathcal{H}$ are equal to their respective sources. Therefore, the projection of the application of $c2_u$ to a state of $\mathcal{C}$ is equivalent to the application of $h1_u$ to the projection of this state. Therefore the projection of σ_C is a sequence of states of $\mathcal{H}$ such that each consequent state is produced by an application of some $h1$ to the previous state (Recall that finite stuttering is eliminated by the definition of a projection). Therefore, to prove that the projection of σ_C is a computation of $\mathcal{H}$ we need to show that this projection is maximal and fair.

There can be two cases. If σ_C is finite, by Theorem 4.2 the projection of this computation ends in a fixpoint. Therefore, the projection is maximal. Since any finite computation is fair, this projection is a computation of σ_H.

If σ_C is infinite, by Theorem 4.2 the projection of this computation cannot end in a fixpoint. Lemma 4.3 implies that this computation is going to be fair. ■

4.3.3. Completeness

To prove completeness we construct a computation of $\mathcal{C}$ that is equivalent to an arbitrary computation of $\mathcal{H}$.

We call a state s of $\mathcal{C}$ *clean* if for any process P_u, $ready_u$ is cleared and the only guard that is possibly enabled at s is $c6_u$. Let $u < v$. In a clean state only $c6_u$ be enabled in P_u. Thus the following should also be true in every clean state:

- the token is held by $a_u.v$, that is: $a_u.v = b_v.u = c_v.u = d_u.v$.
- since $ready_v$ is cleared, $r_u.v$ and $y_u.v$ are also cleared.
- $x_u.v = x_v$.
- $\texttt{request}_u$ is cleared.

THEOREM 4.4 (Completeness). For every computation σ_H there exists a computation of σ_C the projection of which is σ_H.

Proof. Let $s_0, s_1, s_2, \ldots$ be σ_H. We prove the theorem by constructing σ_C such that the projection of σ_C is σ_H.

Let t_0 be a clean state of σ_C such that for every P_u the value of x_u is the same as in s_0. Thus the projection of t_0 is s_0. In a clean state the value of $x_u.v$ is the same as the source x_v. Therefore, if $g_u.k$ is enabled in t_0, it also evaluates to true in s_0. This means that $c6_u$ is enabled in t_0. The execution of $c6_u$ sets $\mathtt{request}_u$ and, therefore, enables $c1_u$.

Let s_1 be a state produced by executing $g_u.k$ at s_0. The execution of $c1_u$ at t_0 increments $a_u.v$ for every neighbor P_v such that $u < v$. Thus $a_u.v$ relinquishes the token. If $a_u.v$ does not have the token there is an enabled GC such that the execution of this GC passes the token further until $a_u.v$ re-acquires the token.

Let us assume that after t_1, σ_C contains the sequence of states such that every state of this sequence is produced by executing a GC that passes the token as described in the previous paragraph. This sequence ends in a state t_i where P_v has $ready_u$ set and for every neighbor P_v, $a_u.v = d_u.v$. Furthermore, $r_v.u$ is cleared for every neighbor P_v. Thus $c2_u$ is enabled at t_i.

Note that the sequence of states $t_0, \ldots, t_i$ does not execute any GC of $\mathcal{H}$. Therefore the projection of this sequence produces just one state - s_0.

Let t_{i+1} be the state produced by executing $c2_u$ at t_i. The execution of $c2_u$ at t_{i+1} increments $a_u.v$ for every neighbor P_v such that $u > v$. Similar to the argument above we can construct a sequence which leads to the state t_j where $a_u.v = d_u.v$ for all neighbor P_v. Note that $t_{i+1}, \ldots, t_j$ does not execute any GC of $\mathcal{H}$. Therefore the projection of this sequence onto $\mathcal{H}$ produces just one state - s_1.

Note also that t_j is a clean state. Similarly we can attach a sequence of states to σ_C that projects to s_2. Continuing in this manner we can construct a sequence of states σ_C such that the projection of it produces σ_H. If σ_H is finite then σ_C ends in a clean state where no $c6$ is enabled (*i.e.* a fixpoint). If σ_H is infinite, so is σ_C.

It remains to be proven that σ_C is fair. Note that all GC that got enabled between t_0 and t_j were executed except for the GCs that correspond to GCs enabled in s_0. Note also that the GC that changes s_0 into s_1 was executed between t_0 and t_j. Since σ_H is fair and all enabled GCs are eventually executed, σ_C is also fair. ∎

5. PERFORMANCE EVALUATION AND COMPARISON WITH OTHER ATOMICITY REFINEMENTS

We define two performance metrics for atomicity refinement, inspired by Singhal and Shivaratri's metrics for synchronization programs [19]. *Step complexity* is the average number of low-atomicity GCs a process executes to simulate one high-atomicity GC. Step complexity depends on the number of simultaneously enabled high-atomicity GCs in program states. We consider two extreme cases: heavy load - when all GCs are always enabled; and light load - only one GC is enabled in the system.

TABLE 1

Performance of different atomicity refinements

refinement authors	step complexity		synchronization delay
	light load	heavy load	
Nesterenko and Arora	$O(n)$	$O(n)$	$O(1)$
Gouda and Haddix	$O(n*d*p)$	$O(n*d)$	$O(d)$
Antonoiu and Srimani	$O(n*p+n*p/M)$	$O(n+n/M)$	$O(d)$

where n - degree of a process, d - diameter of the system, p - number of processes in the system, M - bound on variables ($M \gg n$)

Causally related GCs are commands that execute in order, the execution of one GC enables the next one. *Synchronization delay* is the average number of causally related low-atomicity GCs that are executed between two high-atomicity GCs of two neighbor processes.

Step complexity indicates the amount of computation the refined program has to perform to execute the simulated high-atomicity program. Synchronization delay indicates the speed with which the refinement executes this program.

Throughout this section, let n be the maximum degree of a process (the maximum number of neighbors a process has), d be the diameter of the program (the number the processes in the longest simple path), and p be the number of processes.

To execute a GC of $\mathcal{H}$ a process of $\mathcal{C}$ first executes $c6$; the process circulates the tokens to the higher id neighbors by executing $c1$. The process has to collect back the tokens from the neighbors. For that the neighbors have to execute $c4$, $c3$ and then the process has to execute $c4$ for every higher id neighbor. After the execution, the process communicates with its lower id neighbors in a similar manner. Note that no other GCs (except for possibly $c4$ and $c5$) are executed. Thus, step complexity of our refinement is in $O(n)$. It is independent of the load. The synchronization delay is in $O(1)$.

Gouda and Haddix [12] propose a solution to semantics refinement problem. They consider the model where a process can atomically access the state of all its neighbors and update its own state, and also the model where a process can atomically access the state of one neighbor and update its own state. Even the latter model has higher atomicity than our low-atomicity model. Nevertheless, we compare the performance of the atomicity refinement of Gouda and Haddix with our refinement.

In [12] each process maintains a variable v whose range is proportional to the size of the largest cycle in the system. That is the range of v is proportional to the diameter of the system. Each process continuously increments v and can execute the high-atomicity GC only when v assumes one particular value out of the whole range. The values of these variables in all processes of the system are tightly synchronized: a process cannot proceed to increment its v while the neighbors have not incremented their vs. As it is the case with our refinement, each process has to examine the state of its neighbors at least once before executing a high-atomicity GC. Thus, the step complexity of this program is in $O(d*n*p)$ if the load is light and in $O(d * n)$ if the load is heavy. The synchronization delay of this refinement is in $O(d)$.

Antonoiu and Srimani [1] propose atomicity refinement from high-atomicity model to message-passing system model. They implicitly assume that the implementation of low-atomicity model on message-passing systems has stabilized. Therefore, their refinement essentially provides a transformation from our high-atomicity model to our low-atomicity model.

In [1], the priority of CS access between neighbors is based on bounded timestamps. Timestamps are just integers held in timestamp counters (process ids are used for tie-breaking). A process with the lower timestamp is allowed to enter CS. Every time the process requests CS it is assigned a greater timestamp. This preserves fairness of CS access. The timestamps are bounded and when the timestamp counter reaches the greatest possible value M it is reset to 0. This violates a well-founded order of timestamps and CS access is prohibited when the system has timestamps that are close to the maximum value. Antonoiu and Srimani show that the range of the timestamps where CS access is disabled is proportional to p. Therefore, M has to be far greater than p for their program to be efficient. The control of the spread between the timestamps in the system is based on the use of a spanning tree. A process can increment its timestamp (and request CS) only if the timestamps of its children and the parent are close. On average, to execute a GC of $\mathcal{H}$ each process in the system has to increment its timestamp. Therefore, step complexity of this refinement is in $O(n * p + n * p/M)$ under light load and in $O(n + n/M)$ under heavy load. To execute a GC of $\mathcal{H}$, by some process P_i the timestamps of all processes on the longest path from the root to leaves containing P_i have to be incremented in order. Thus the synchronization delay of the refinement is in $O(d)$

The refinements in [1, 12] are not fixpoint preserving. Also, these refinements require tight synchronization between processes and do not allow certain computations of $\mathcal{H}$; in particular a process cannot repeatedly execute its GCs while its neighbor has a GC enabled. Therefore, these refinements are not complete.

6. EXTENSIONS AND CONCLUDING REMARKS

In this paper, we presented a technique for stabilization-preserving atomicity refinement of concurrent programs. Our refinement enables design of stabilizing programs in a simple but restrictive model and implementation in a more complex but efficient model. It is based on a stabilizing, bounded-space, dining philosopher program in the more complex model. Moreover, it is not only sound but also complete, and is fixpoint- and fairness-preserving.

In conclusion, we discuss several notable extensions of our refinement.

6.1. Semantics Refinement

Consider the semantics refinement problem where the abstract model uses interleaving semantics, the concrete model uses power-set semantics and both models employ high-atomicity guarded commands. We show that our refinement solves this problem. Note that low-atomicity actions are just restricted high-atomicity actions and, hence, a low-atomicity program is a high-atomicity program. Thus, it suffices to show that for a low-atomicity program a power-set computation is equivalent to an interleaving computation.

Two computations are *equivalent* if in both computations every P_u executes the same sequence of GCs and when a GC executes the values of the variables it reads are the same. Recall that in a computation under interleaving semantics (interleaving computation) each consequent state is produced by the execution of one of the GC that is enabled in the preceding state. In a computation under power-set semantics (power-set computation) each consequent state is produced by the execution of any number of GCs that is enabled in the preceding state.

THEOREM 6.1. For every power-set computation of a low-atomicity program there is an equivalent interleaving computation.

Proof. It suffices to show that for every pair of consequent states (s_1, s_2) of power-set computation there is an equivalent sequence of states of interleaving computation. The GCs executed in s_1 are either synchs or updates. Clearly, if the synchs are executed one after another and followed by the updates the resulting interleaving sequence is equivalent to the pair (s_1, s_2). ■

6.2. Generalization to Drinking Philosophers Problem

$\mathcal{DP}$ can be extended to solve drinking philosophers problem [7], to increase the efficiency of our refinement. In the argument below we assume that each process of $\mathcal{H}$ has multiple GCs. Two GCs in neighbor processes in $\mathcal{H}$ *conflict* (affect each other) if one of them writes (updates) the variables the other GC reads. Loss of concurrency occurs in $\mathcal{DP}$ since it enforces MX of execution of GCs of neighbor processes, regardless of whether these GCs actually conflict.

Recall that to ensure MX among neighbor processes in $\mathcal{DP}$ every pair of neighbors P_u and P_v maintains a sequence of handshake variables H_{uv}. Sending a token along this handshake sequence is used to inform the neighbor if the process is entering or exiting CS. Similarly, P_u and P_v can be given a sequence of handshake variables for every pair of conflicting guarded commands. Then if a GC of $\mathcal{H}$ is enabled the tokens are sent along each sequence to prevent the conflicting guarded commands from executing concurrently. More precisely: the tokens are sent to the processes with higher ids before the GC is executed and to the processes with smaller ids afterwards. Thus, if two GCs of $\mathcal{H}$ on neighbor processes do not conflict no synchronization between neighbors is done.

We would like to elaborate on how the increase of concurrency in the dining philosophers generalization relates to the completeness property for the original dining philosopher program. Even though the completeness result states that for every high-atomicity computation there is a low-atomicity one whose projection is this high-atomicity computation. Note, however, that the low-atomicity steps are removed when the projection is taken. Thus, the completeness result does not take into account the number of low-atomicity steps each high-atomicity step execution requires. This number is in general smaller for the drinking philosophers generalization of our dining philosophers solution. In practice it would result in higher concurrency of high-atomicity GC execution of the low-atomicity program.

6.3. Modification to Solve Fairness Refinement

Our refinement is readily modified to relax the assumption of weak-fairness in the concrete model. For this we need to change $\mathcal{DP}$ to operate correctly without any fairness. In principle, relaxing fairness assumptions does not violate the safety properties of $\mathcal{DP}$ but care has to taken to ensure that it does not invalidate the liveness properties.

The modifications are as follows. We remove `request` from the GC of $dp1$, so every process continuously requests to enter CS. The command of $dp3$ is added to the command of $dp1$ and the guard of $dp3$ is augmented so that it evaluates to true only if $dp1$ and $dp2$ are not enabled. $dp1$ and $dp2$ are responsible for executing CS of the local process. $dp3$ is used to propagate the token for the neighboring processes.

The above modifications ensure that a process cannot continually enter CS while a neighbor is in CS contention even if no fairness is assumed. Since every process is always in CS contention, every process enters CS by induction on the longest path in the system. Thus fairness is refined.

6.4. Extension to Message-Passing Systems

Our refinement is further extended so that the concrete program executes in message-passing model where the processes communicate via finite capacity lossy channels. Again, $\mathcal{DP}$ has to be modified to to work in this model, as follows. Given that H_{uv} in $\mathcal{DP}$ is used to pass information from process P_u to its neighbor P_v and get an acknowledgment that this information has been received, an alternating-bit protocol (ABP) can be used for the same purpose in message-passing systems. A formal model of dealing with lossy bounded channels in message-passing systems as well as a stabilizing ABP is presented in [14].

In the modified $\mathcal{DP}$, P_u sends the value of a handshake variable (together with the rest of its state) to P_v in a message. If the message is lost it is retransmitted by a timeout. When P_v receives the message it copies the state of P_u (including the handshake variable) into its image variables and sends a reply back to P_u. When P_u gets the reply it knows that P_v got the original message. It is proven [14] that ABP stabilizes when the range of the handshake variables is greater than the sum of the capacity of the channels between P_u and P_v and in the opposite direction.

When $\mathcal{H}$ reaches a fixpoint the values of the variables of processes of $\mathcal{C}$ extended to message passing system do not change. Thus $\mathcal{C}$ is in a quiescent state. But, as is well-known [13] that a stabilizing message-passing program cannot reach a fixpoint, the extension of $\mathcal{DP}$ to message-passing systems is no longer fixpoint-preserving: the timeout has to be executed even if the projection of the program has reached a fixpoint.

APPENDIX

A.1. RING STABILIZATION

THEOREM A.1. The disjunction of the following predicates is stabilizing for $\mathcal{DP}$.

$$
\begin{aligned}
&\forall (P_u, P_v) \in N: \\
&(a_u.v = b_v.u = c_v.u = d_v.u) \vee && (R_9)\\
&(a_u.v \neq b_v.u = c_v.u = d_v.u) \vee && (R_{10})\\
&(a_u.v = b_v.u \neq c_v.u = d_v.u) \vee && (R_{11})\\
&(a_u.v = b_v.u = c_v.u \neq d_v.u) && (R_{12})
\end{aligned}
$$

Proof. We show closure first. Out of ten GCs of P_u and P_v only $dp1_u$, $dp2_u$, $dp4_v$, $dp3_v$, and $dp4_u$ can affect the predicates. When R_9 holds, only $dp1_u$ or $dp2_u$ can be enabled (depending on whether v is greater or smaller than u). When R_{10}, R_{11}, R_{12} hold only $dp4_v$, $dp3_v$ and $dp4_u$ are enabled respectively. Note that the execution of the enabled GC brings the program to a state where one of the predicates holds.

We demonstrate convergence by reduction of the program to Dijkstra K-state token circulation algorithm [10]. The difference between our handshake sequence and Dijkstra's algorithm is that $dp1_u$ (or $dp2_u$) may or may not be enabled when $a_u.v = d_v.u$. If the execution of the $dp1_u$ (or $dp2_u$) is weakly fair wrt $a_u.v = d_v.u$ in some computation σ_C. Then H_{uv} behaves like Dijkstra's algorithm and stabilizes. If execution of the $dp1_u$ (or $dp2_u$) is not weakly fair wrt $a_u.v = d_v.u$ in $a_u.v = d_v.u$ then there is an infinite suffix of the computation where $a_u.v = d_v.u$ and $a_u.v$ is not incremented. Clearly then, the program stabilizes to R_9. ■

A.2. ACRONYMS AND NOTATION

MX	mutual exclusion problem
DP	dining philosophers problem
CS	critical section
GC	guarded command
ABP	alternating-bit protocol

$\mathcal{P}$, $\mathcal{Q}$	programs
$\mathcal{DP}$	dining philosophers program
$\mathcal{H}$	high-atomicity program
$\mathcal{C}$	refined program
P	process
N	neighbor relation between processes
i, j, k, u, v	process identifiers
P_u, P_v	neighbor processes
R, S	predicates
a, b, c, d	handshake variables
H_{uv}	sequence of handshake variables between P_u and P_v
s, t	program states

$I_{\mathcal{DP}}$	invariant of $\mathcal{DP}$
$I_{\mathcal{C}}$	invariant of $\mathcal{C}$
σ_H	computation of $\mathcal{H}$
σ_C	computation of $\mathcal{C}$
n	maximum degree of a process
d	diameter of the system
p	number of processes
M	timestamp bound

REFERENCES

1. G. Antonoiu and P.K. Srimani. Mutual exclusion between neighboring nodes in an arbitrary system graph that stabilizes using read/write atomicity. In *Proceedings of EuroPar'99*, volume 1685 of *Lecture Notes in Computer Science*, pages 823–830. Springer-Verlag, 1999.
2. A. Arora and M.G. Gouda. Closure and convergence: a foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19:1015–1027, 1993.
3. A. Arora and M.G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43:1026–1038, 1994.
4. B. Awerbuch, B. Patt-Shamir, and G. Varghese. Bounding the unbounded (distributed computing protocols). In *Proceedings IEEE INFOCOM 94 The Conference on Computer Communications*, pages 776–783, 1994.
5. G.M. Brown, M.G. Gouda, and C.L. Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, 38:845–852, 1989.
6. J.E. Burns. Self-stabilizing rings without demons. Technical Report GIT-ICS-87/36, Georgia Tech, 1987.
7. K.M. Chandy and J. Misra. The drinking philosopher's problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.
8. K.M. Chandy and J. Misra. *Parallel Program Design : a Foundation.* Addison-Wesley, Reading, Mass., 1988.
9. E.W. Dijkstra. *Hierarchical Ordering of Sequential Processes*, pages 72–93. Academic Press, 1972.
10. E.W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.
11. S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.
12. M.G. Gouda and F. Haddix. The alternator. submitted to Journal of Parallel and Distoributed Computing.
13. M.G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448–458, 1991.
14. R.R. Howell, M. Nesterenko, and M. Mizuno. Finite-state self-stabilizing protocols in message passing systems. In *Proceedings of the Fourth Workshop on Self-Stabilizing systems*, pages 62–69, 1999.
15. L. Lamport. The mutual exclusion problem: part ii-statement and solutions. *Journal of the Association of the Computing Machinery*, 33:327–348, 1986.
16. M. Mizuno and H. Kakugawa. A timestamp based transformation of self-stabilizing programs for distributed computing environments. In *WDAG96 Distributed Algorithms 10th International Workshop Proceedings, Springer-Verlag LNCS:1151*, pages 304–321, 1996.
17. M. Mizuno and M. Nesterenko. A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. *Information Processing Letters*, 66(6):285–290, 1998.
18. M. Mizuno, M. Nesterenko, and H. Kakugawa. Lock-based self-stabilizing distributed mutual exclusion algorithms. In *Proceedings of the Sixteenth International Conference on Distributed Computing Systems*, pages 708–716, 1996.
19. M. Singhal and N.G. Shivaratri. *Advanced Concepts in Operating Systems: dsitributed, database, and multiprocessor operating systems.* McGraw-Hill, 1994.